Constructing Regular Expressions from Real-life Event Logs

No Author Given

No Institute Given

Abstract. Process mining is a new discipline aimed at constructing process models from event logs. Recently several methods for the discovery of transition systems from event logs were introduced. Considering these transition systems as finite state machines classical algorithms for deriving regular expressions can be applied. Regular expressions allow representing sequential process models in a hierarchical way, using sequence, choice, and iterative patterns. The aim of this work is to apply and tune an algorithm deriving regular expressions from transition systems within the process mining domain.

1 Introduction

Process mining is a research field [1] providing methods for the discovery of process models from event logs. *Representational bias*, i.e., a set of restrictions imposed on target process models, plays an important role in process mining. Compact and structured process models clearly represent the underlying processes. One of the aims of the process discovery algorithms is to construct such models from real-life event data. Powerful and widely-used methods for the discovery of process trees (structured models based on sequential, choice, parallel, and iterative patterns) were proposed in [2, 3]. Usually, these methods are used under an assumption that not all the traces (sequences of events) of the event log will fit the target process model. In the opposite case, the resulting process model may be too general allowing all the possible behavior. In this work, we assume that the analyst may want to discover fitting and precise structured process models that exactly represent the process behavior.

Regular expressions, which are used to model sequential processes with choice, sequence, and iterative patterns, were selected as a representation bias within this work. The technique proposed in this paper consists of two steps: (1) a transition system is constructed from an event log, using an existing transition system discovery technique [4]; from now on we will call these transitions systems *finite state machines*; (2) a regular expression is synthesized from the given finite state machine. There are several algorithms for constructing regular expressions from finite state machines [5–7]. All these algorithms preserve the language, i.e., finite state machines and resulting regular expressions accept the same set of activity sequences. In this paper we use the algorithm based on the reduction of

states [7]. Additionally, we extend this algorithm and use heuristics based on the particular properties of the finite state machines constructed from event logs. This algorithm was implemented and tested on real-life event logs showing its capability to construct compact regular expressions in comparison to the regular expressions constructed without the proposed heuristics.

2 Preliminaries

In this section we present basic notions used throughout the paper. Let $\mathcal{A}$ be a set of activities. Let $\mathcal{A}^*$ be a set of all possible sequences of activities. *Language* $\mathcal{L}$ is a set of sequences of activities, i.e., $\mathcal{L} \subseteq \mathcal{A}^*$.

The *regular language* is a language over the set activities $\mathcal{A}$ iff it can be constructed inductively based on the following rules: (1) $\emptyset$ is a regular language; (2) $\{\langle\rangle\}$ is a regular language, denoted as ϵ; (3) $\{\langle a_i\rangle\}$, where $a_i \in \mathcal{A}$, is a regular language, denoted as a_i; (4) let $P, Q \subseteq \mathcal{A}^*$ be regular languages, then (4.1) $P.Q = \{\langle p_1, ..., p_n, q_1, ..., q_m\rangle | \langle p_1, ..., p_n\rangle \in P, \langle q_1, ..., q_m\rangle \in Q\}$ is a regular language, (4.2) $P|Q = \{\sigma | \sigma \in P \cup Q\}$ is a regular language, and (4.3) $P^* = \cup_{i=0}^{\infty} P^n$, where $P^0 = \{\langle\rangle\}$, $P^1 = P$, and $P^n = P^{n-1}.P$, if $n > 1$, is a regular language.

A *finite state machine* is a tuple $FSM = (S, E, T, s_i, S_f)$, where S is a finite set of *states*, E is a finite set of *events* (*activities*), $T \subseteq (S \times E \times S)$ is a set of *transitions*, $s_i \in S$ is an *initial state*, and $S_f \subseteq S$ is a set of *accepting states*. Sequence $t = \langle a_1, ..., a_n\rangle \in \mathcal{A}^*$ is *accepted* by a finite state machine $FSM = (S, \mathcal{A}, T, s_i, S_f)$ iff there exists a sequence of transitions (s_i, a_1, s_1),(s_1, a_2, s_2),...,(s_{n-1}, a_n, s_n), leading from the initial state s_i to an accepting state $s_n \in S_f$. *Language* of *FSM*, denoted by $\mathcal{L}(FSM)$, is a set of all accepted sequences of *FSM*.

3 Algorithm description

This section presents an algorithm for constructing regular expressions from event logs. The approach presented in this paper consists of two steps: (1) a finite state machine is constructed from an event log, (2) a regular expression is synthesized from the given finite state machine (Fig. 2).

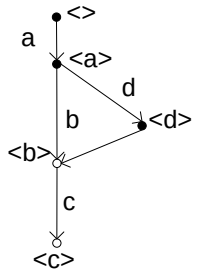


Fig. 1. A finite state machine constructed from the event log E.

In this work we used a method for constructing finite state machines (transition systems) from event logs [4] based on the prefixes of length 1. Let us demonstrate this approach by an example. Since each event log is a set of traces or, in other words, sequences of activities, it represents a language. Consider an event log $E = \{\langle a, b\rangle, \langle a, b, c\rangle, \langle a, d, b\rangle\}$. An example of a finite state machine constructed from this event log is presented in Fig. 1. Accepting states are shown in white and each state corresponds to a sequence of last traversed activities of the length less or equal 1. Although in this concrete example the event log and

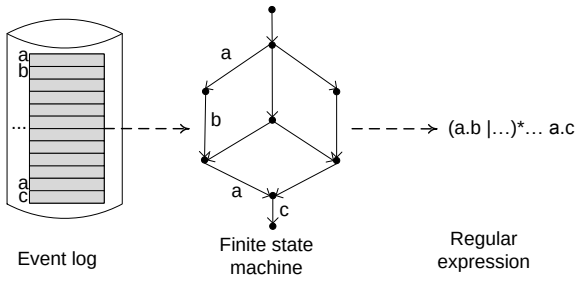


Fig. 2. Discovery approach.

the language of the corresponding finite state machine coincide, in the general case it is guaranteed that the event log is a subset of the language of the corresponding finite state machine.

In the second step, we apply the approach based on the removal of states to construct regular expressions from finite state machines [7]. The main idea of the state removal approach is to assume that transitions can be labeled not by activities, but by regular expressions. This algorithm starts with the initial finite state machine and then inductively remove the states in order to obtain resulting regular expressions (Fig. 3). At the final steps of the algorithm these resulting expressions label the transitions which lead from the initial state to an accepting state. The order of the processed states significantly affects the size (the length) of the resulting regular expression. We used the strategy proposed in [8]. According to this strategy, the next state which should be removed is a state that produces minimal regular expressions. Thus, the state s with the minimal value $W(s) = (l-1)\sum_{i=1}^{m}|r_{s_i s}| + (m-1)\sum_{i=1}^{l}|r_{s s_i}| + (ml-1)|r_{ss}|$, where m (l) is the number of incoming (outgoing) transitions for s, and $r_{s_i s_j}$ is a regular expression marking transition from s_i to s_j, will be removed first.

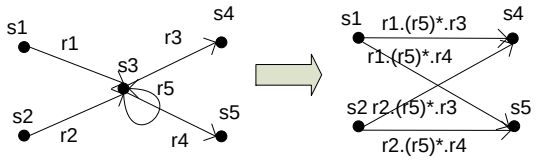


Fig. 3. State removal technique.

Basing on special characteristics of the finite state machines constructed from event logs we propose heuristics aimed to reduce and structure the resulting regular expressions. Let us consider a fragment of a finite state machine presented in Fig. 4. Since this finite state machine was constructed using prefixes of length 1, each state corresponds to a concrete activity and all incoming transitions are marked with this activity. After the removal of state s there will be

three regular expressions $a.b^*.c$ and three regular expressions $a.b^*.d$. A repeated regular expression can be replaced by a single symbol denoting a subprocess. This replacement can be applied not only in the first step of the state removal algorithm, but also in the next steps, because (it is confirmed experimentally in the next section) incoming transitions of a state can be labeled by identical regular expressions. Additionally, if incoming transitions are not labeled by identical regular expressions, prefixes, such as $a.b^*$ (for the case presented in Fig. 3), can also form subprocesses.

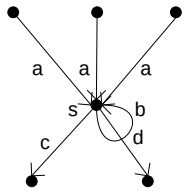


Fig. 4. A fragment of a finite state machine.

4 Experimental results

The state reduction algorithm including the ordering (presented in [8]) and the prefix-based (proposed in this paper) heuristics was implemented and tested on real-life event logs.

Three real-life event logs of a bug tracking system (TS), banking financial system (FS) and flight booking system (BS) were analyzed. First, all these event logs were loaded to ProM (Process Mining Framework) [9] and corresponding finite state machines were constructed. Then these finite state machines were loaded to the tool specially developed by the authors in order to synthesize regular expressions.

Table 1 shows the length of regular expressions constructed without the prefix-based heuristics proposed in this paper, as well as the size (length, length of the root process, average length of subprocesses, depth) of regular expressions constructed with the prefix-based heuristics.

Event log	**Number of states**	**Length of RE**	**Length of RE with replacement (root + subprocesses)**	**Length of the root process**	**Average length of subprocesses**	**Maximal depth**
TS	5	48	39	16	2.75	2
FS	7	22	21	14	2.33	1
FS	9	66	54	32	2.2	2
FS	12	397	175	99	3.8	3
FS	13	747	207	97	3.93	5
FS	19	2480	490	214	6.27	6
BS	5	19	16	11	2.5	2
BS	7	569	106	16	5.625	4
BS	9	11000	337	114	5.87	7
BS	11	41953	643	135	9.24	8

Table 1. The size of the synthesized regular expressions.

As it follows from the experiment results, the prefix-based heuristics can significantly help to reduce and structure the resulting regular expression. The length of the root regular expression is usually significantly lower than the length of the corresponding regular expression constructed without applying the prefix-based heuristics. Additionally, the proposed approach allows us to extract repeating regular expressions (subprocesses), providing information about repeating patterns within the event log.

Now let us consider the event log of a bug tracking system (TS) in details. This log is defined over five activities of a bug tracking process: a (*"In Progress"*), b (*"Resolved"*), c (*"Closed"*), d (*"Reopened"*), and e (*"Open"*). The regular expression constructed without the prefix-based heuristics is the following: $c|a.(a)^*|c|b|a.(a)^*.b|c|(e|c.d|a.(a)^*.e|c.d|b|a.(a)^*.b.(d|c.d)).(c.d|a.(a)^*.e|c.d|b|a.(a)^*.b.(d|c.d))^*.(\epsilon|c|a.(a)^*|c|b|a.(a)^*.b|c)$.

The root regular expression constructed using the proposed heuristics is: $c|F|H|(B|\ E|D.(d|C)).(C|E|D.(d|C))^*.(A|F|H)$. The subprocesses are defined in Tab. 2.

Subprocess	**Definition**
A	$\epsilon\|c$
B	$e\|c.d$
C	$c.d$
D	$b\|a.(a)^*.b$
E	$a.(a)^*.B$
F	$a.(a)^*.A$
G	$D.(d\|C)$
H	$D.A$

Table 2. Subprocesses of the bug tracking process.

This example shows that the regular expression constructed using the proposed heuristics is more structured and can be easy analyzed. For example, one may note there are four main patterns of behavior: c (only *"Closed"* activity), F (repeating *"In Progress"* and then may be *"Closed"*), H (zero or more times *"In Progress"*, then *"Resolved"* and then may be *"Closed"*), and $(B|E|D.(d|C)).(C|E|D.(d|C))^*.(A|F|H)$. The last pattern can also be easy analyzed, it contains three explicit parts corresponding to the begging, main execution and termination of the process. All this analysis is not possible with the first regular expression, constructed without the prefix-based heuristics.

5 Conclusion

In this paper we consider an approach for constructing regular expressions from event logs. The main goal of this research is to assess the possibility of applying the regular expression synthesis techniques within process mining field. The proposed approach consists of two steps: first, we construct a finite state machine for an event log using existing techniques, then a regular expression is

derived from this finite state machine. For constructing regular expressions the state removal method was selected. Additionally, heuristics aimed at reducing the size of the resulting regular expressions were applied. Besides that, special heuristics applicable for certain type of finite state machines constructed from event logs were proposed. The presented approach was implemented and tested on real-life event logs. It was shown that the proposed heuristics allow us to significantly reduce the size and improve the structure of the resulting regular expressions. In future we plan to tackle other types of finite state machines (not only based on prefixes of size 1) constructed from event logs.

References

1. van der Aalst, W.M.P.: Process Mining: Data Science in Action. 2 edn. Springer, Heidelberg (2016)
2. Leemans, S., Fahland, D., van der Aalst, W.: Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In Colom, J., Desel, J., eds.: Application and Theory of Petri Nets and Concurrency: 34th International Conference, PETRI NETS 2013, Milan, Italy. Volume 7927 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 311–329
3. Leemans, S., Fahland, D., van der Aalst, W.: Discovering Block-Structured Process Models from Incomplete Event Logs. In Ciardo, G., Kindler, E., eds.: Application and Theory of Petri Nets and Concurrency: 35th International Conference, PETRI NETS 2014, Tunis, Tunisia. Volume 8489 of Lecture Notes in Computer Science. Springer International Publishing, Cham (2014) 91–110
4. van der Aalst, W.M.P., Rubin, V., Verbeek, H.M.W., van Dongen, B.F., Kindler, E., Günther, C.W.: Process mining: a two-step approach to balance between underfitting and overfitting. Software & Systems Modeling **9**(1) (Nov 2008) 87
5. Kleene, S.C.: Representation of events in nerve nets and finite automata. In Shannon, C., McCarthy, J., eds.: Automata Studies. Princeton University Press, Princeton, NJ (1956) 3–41
6. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4) (October 1964) 481–494
7. Linz, P.: An Introduction to Formal Language and Automata. Jones and Bartlett Publishers, Inc., USA (2006)
8. Delgado, M., Morais, J.: Approximation to the smallest regular expression for a given regular language. In Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S., eds.: Implementation and Application of Automata, Berlin, Heidelberg, Springer Berlin Heidelberg (2005) 312–314
9. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The prom framework: A new era in process mining tool support. In Ciardo, G., Darondeau, P., eds.: Applications and Theory of Petri Nets 2005, Berlin, Heidelberg, Springer Berlin Heidelberg (2005) 444–454